3DGS-LM: Faster Gaussian-Splatting Optimization with Levenberg-Marquardt

Supplementary Material

A. More Details About CUDA Kernel Design

We introduce the necessary CUDA kernels to calculate the PCG algorithm in Sec. 3.3. In this section, we provide additional implementation details.

A.1. Parallelization Pattern

We implement the *per-pixel-per-splat* parallelization pattern in our CUDA kernels by reading subsequent entries from the gradient cache in subsequent threads. This makes reading cache values perfectly coalesced and therefore minimizes the overhead caused by the operation. The gradient cache is sorted over Gaussians, which means that subsequent entries refer to different rays (pixels) that saw this projected Gaussian (splat). In general, one thread handles all residuals corresponding to the respective pixel, i.e., many computations are shared across color channels and are therefore combined.

A.2. Design Of `buildCache` And `applyJT`

The necessary computations for the `buildCache` and `applyJT` steps in PCG (see Algorithm 1) are split across three kernels. This follows the original design of the 3DGS differentiable rasterizer [23]. In both cases, we calculate the Jacobian-vector product of the form $\mathbf{g}{=}\mathbf{J}^T\mathbf{u}$ where $\mathbf{J}{\in}\mathbb{R}^{NxM}$ is the Jacobian matrix of N residuals and M Gaussian parameters and $\mathbf{u}{\in}\mathbb{R}^N$ is an input vector. The k-th element in the output vector is calculated as:

$$\mathbf{g}_k = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}\mathbf{u}_i = \frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k}\sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\mathbf{u}_i \tag{9}$$

where $\mathbf{r}_i$ is the i-th residual and $\mathbf{x}_k$ is the k-th Gaussian parameter. In other words, $\mathbf{g}_k$ is a sum over all residuals for the k-th Gaussian parameter. Following the chain-rule, it is possible to split up the gradient $\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}{=}\frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k}$. We can use this to split the computation across three smaller kernels, where only the first needs to calculate the sum over all residuals: $\sum_{i=0}^{N}\frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\mathbf{u}_i$. This sum is the main bottleneck for the kernel implementation, since it needs to be implemented atomically (i.e., multiple threads write to the same output position). The other kernels then only calculate the remaining steps by parallelizing over Gaussians. We slightly abuse notation and denote with $\mathbf{y}_k$ the 2D mean, color, and opacity attributes of the k-th projected Gaussian. That is, the first kernel sums up the partial derivatives to each of these attributes in separate vectors. In the following, we add the suffixes `_p1, _p2, _p3` to denote the three kernels for the respective operation (where `_p1` refers to the kernel that calculates the sum over residuals).

The `buildCache_p1` utilizes the original *per-pixel* parallelization, whereas the `applyJT_p1` kernel use the gradient cache and our proposed *per-pixel-per-splat* parallelization pattern. The gradient cache is sorted over Gaussians, i.e., subsequent entries correspond to different rays of the same splat. This allows us to efficiently implement the sum by first performing a segmented warp reduce and then only issuing one `atomicAdd` statement per warp.

A.3. Design Of `applyJ` And `diagJTJ`

In contrast, the `applyJ` and `diagJTJ` computations cannot be split up into smaller kernels. Concretely, the `applyJ` kernel calculates $\mathbf{u}{=}\mathbf{J}\mathbf{p}$ with $\mathbf{p}{\in}\mathbb{R}^M$. The k-th element in the output vector is calculated as:

$$\mathbf{u}_k = \sum_{i=0}^{M} \frac{\partial \mathbf{r}_k}{\partial \mathbf{x}_i}\mathbf{p}_i = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_k}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \mathbf{x}_i}\mathbf{p}_i \tag{10}$$

In other words, $\mathbf{u}_k$ is a sum over all Gaussian attributes for the k-th residual. Similarly, the `diagJTJ` kernel calculates $\mathbf{M} = \text{diag}(\mathbf{J}^T\mathbf{J}) \in \mathbb{R}^M$. The k-th element in the output vector is calculated as:

$$\mathbf{M}_k = \sum_{i=0}^{N} \left(\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}\right)^2 = \sum_{i=0}^{N} \left(\frac{\partial \mathbf{r}_i}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial \mathbf{x}_k}\right)^2 \tag{11}$$

In both cases it is not possible to move part of the gradients outside of the sum. As a consequence, both `applyJ` and `diagJTJ` are implemented as one kernel, where each thread directly calculates the final partial derivatives to all Gaussian attributes. This slightly increases the number of required registers and the runtime compared to the `applyJT` kernel (see Tab. 5). The `diagJTJ` kernel makes use of the same segmented warp reduce as `applyJT_p1` for efficiently summing up the squared partial derivatives. The `applyJ` kernel first sums up over all Gaussian attributes within each thread separately. Then, we only issue one `atomicAdd` statement for each residual per thread.

The `applyJ` kernel requires the input vector $\mathbf{p}$ to be sorted per Gaussian to make reading from it coalesced. That is: $\mathbf{p}{=}[x_1^a, ..., x_1^z, ..., x_M^a, ..., x_M^z]^T$, where x_k^a is the value corresponding to the a-th parameter of the k-th Gaussian. In total, each Gaussian consists of 59 parameters: 11 for position, rotation, scaling, and opacity and 48 for all Spherical Harmonics coefficients of degree 3. In contrast, all other kernels require the output vector to be sorted per attribute to make writing to it coalesced. That is: $\mathbf{q}{=}[x_1^a, ..., x_M^a, ..., x_1^z, ..., x_M^z]^T$. We use the structure of $\mathbf{q}$ for all other vector-vector calculations in Algorithm 1 as

well. Whenever we call the `applyJ` kernel, we thus first call the `sortX` kernel that restructures $\mathbf{q}$ to the layout of $\mathbf{p}$.

A.4. Precomputation Of Residual-To-Pixel Weights

We adopt the square root formulation of the residuals in our energy formulation (see Eq. (3)). We efficiently precompute the contribution of the square root to the partial derivatives of Eq. (9), Eq. (10), and Eq. (11). In the following, we divide the partial derivatives from the i-th residual to the k-th Gaussian attribute into two stages:

$$\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k} = \frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i} \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k} \tag{12}$$

where $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$ goes from the residual to the rendered pixel color and $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ from the pixel color to the Gaussian attribute. Since we adopt the L1 and SSIM loss terms and take their square root, the terms $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$ need to be calculated accordingly. In contrast, when using the L2 loss they take a trivial form of $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$=1. In the following, we show that we can simplify the calculation of $\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}$ in the kernels by precomputing $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$.

The k-th element of $\mathbf{g}$ is calculated as:

$$\mathbf{g}_k = (\mathbf{J}^T \mathbf{J} \mathbf{p})_k = \sum_{i=0}^{N} \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k} \sum_{j=0}^{M} \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_j} \mathbf{p}_j \tag{13}$$

By substituting Eq. (12) into Eq. (13), we factor out $\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i}$:

$$\mathbf{g}_k = \sum_{i=0}^{N} (\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i})^2 \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k} \sum_{j=0}^{M} \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_j} \mathbf{p}_j \tag{14}$$

The terms $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ are identical for a residual of the same pixel and color channel that corresponds to either the L1 or SSIM loss terms, respectively. To avoid computing $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ twice (and therefore doubling the grid size of all kernels), we instead sum up the contribution of both loss terms:

$$(\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i})^2 = (\frac{\partial \mathbf{r}_i^1}{\partial \mathbf{c}_i})^2 + (\frac{\partial \mathbf{r}_i^2}{\partial \mathbf{c}_i})^2 \tag{15}$$

where $\mathbf{r}_i^1$ corresponds to the i-th L1 residual and $\mathbf{r}_i^2$ to the i-th SSIM residual. Additionally, we implement the multiplication with $(\frac{\partial \mathbf{r}_i}{\partial \mathbf{c}_i})^2$ in Eq. (14) as elementwise vector product (denoted by $\odot$) of $\hat{\mathbf{u}}$=$[\sum_{j=0}^{M} \frac{\partial \mathbf{c}_0}{\partial \mathbf{x}_j} \mathbf{p}_j ... \sum_{j=0}^{M} \frac{\partial \mathbf{c}_N}{\partial \mathbf{x}_j} \mathbf{p}_j]$ and $\nabla \mathbf{r}$=$[(\frac{\partial \mathbf{r}_0}{\partial \mathbf{c}_0})^2 ... (\frac{\partial \mathbf{r}_N}{\partial \mathbf{c}_N})^2]$:

$$\mathbf{g}_k = \sum_{i=0}^{N} \frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k} (\hat{\mathbf{u}} \odot \nabla \mathbf{r})_i \tag{16}$$

This avoids additional uncoalesced global memory reads to $\nabla \mathbf{r}$ in the CUDA kernels. Instead, we calculate $\hat{\mathbf{u}} \odot \nabla \mathbf{r}$ in a separate operation after the `applyJ` kernel and before `applyJT`. This also simplifies the kernels, since they now only need to compute $\frac{\partial \mathbf{c}_i}{\partial \mathbf{x}_k}$ instead of $\frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_k}$. Therefore, the only runtime overhead of using the L1 and SSIM residual terms over the L2 residuals is the computation of $\nabla \mathbf{r}$. However, this can be efficiently computed using backpropagation (autograd) and is therefore not a bottleneck.

B. Derivation Of Image Subsampling Weights

We subsample batches of images to decrease the size of the gradient cache (see Sec. 3.3). To combine the update vectors from multiple batches, we calculate their weighted mean, as detailed in Eq. (8). This weighted mean approximates the "true" solution to the normal equations (Eq. (4)), that does not rely on any image subsampling (and instead uses all available training images). When subsampling images, we split the number of total residuals into smaller chunks. In the following, we consider the case of two chunks (labeled as $_1$ and $_2$), but the same applies to any number of chunks. We re-write the normal equations (without subsampling) using the chunk notation as:

$$\begin{bmatrix} \mathbf{J}_1^T & \mathbf{J}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{J}_1 \\ \mathbf{J}_2 \end{bmatrix} \Delta = \begin{bmatrix} \mathbf{J}_1^T & \mathbf{J}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{F}_1(\mathbf{x}) \\ \mathbf{F}_2(\mathbf{x}) \end{bmatrix} \tag{17}$$

where we drop the additional LM regularization term for clarity and divide the Jacobian and residual vector into separate matrices/vectors according to the chunks. The solution to the normal equations is obtained by:

$$\Delta = (\mathbf{J}_1^T \mathbf{J}_1 + \mathbf{J}_2^T \mathbf{J}_2)^{-1} (\mathbf{J}_1^T \mathbf{F}_1(\mathbf{x}) + \mathbf{J}_2^T \mathbf{F}_2(\mathbf{x})) \tag{18}$$

In contrast, when we subsample images, we solve the normal equations separately and obtain two solutions:

$$\Delta_1 = (\mathbf{J}_1^T \mathbf{J}_1)^{-1} \mathbf{J}_1^T \mathbf{F}_1(\mathbf{x}) \tag{19}$$

$$\Delta_2 = (\mathbf{J}_2^T \mathbf{J}_2)^{-1} \mathbf{J}_2^T \mathbf{F}_2(\mathbf{x}) \tag{20}$$

We can rewrite Eq. (18) as a weighted mean of Δ_1, Δ_2:

$$\Delta = K^{-1} (\mathbf{J}_1^T \mathbf{J}_1)(\mathbf{J}_1^T \mathbf{J}_1)^{-1} (\mathbf{J}_1^T \mathbf{F}_1(\mathbf{x})) \tag{21}$$

$$+K^{-1} (\mathbf{J}_2^T \mathbf{J}_2)(\mathbf{J}_2^T \mathbf{J}_2)^{-1} (\mathbf{J}_2^T \mathbf{F}_2(\mathbf{x})) \tag{22}$$

$$= w_1 \Delta_1 + w_2 \Delta_2 \tag{23}$$

where $K = (\mathbf{J}_1^T \mathbf{J}_1 + \mathbf{J}_2^T \mathbf{J}_2)$, $w_1 = K^{-1}(\mathbf{J}_1^T \mathbf{J}_1)$, $w_2 = K^{-1}(\mathbf{J}_2^T \mathbf{J}_2)$. Calculating these weights requires to materialize and invert K, which is too costly to fit in memory. To this end, we approximate the true weights w_1 and w_2 with $\tilde{w}_1 = \text{diag}(w_1)$ and $\tilde{w}_2 = \text{diag}(w_2)$. This directly leads to the weighted mean that we employ in Eq. (8).

Kernel	Runtime (ms) ↓	Compute Throughput (%)↑	Memory Throughput (%)↑	Register Count ↓
buildCache_p1	31.32	78.56	78.56	64
buildCache_p2	0.53	17.43	87.94	58
buildCache_p3	4.12	4.54	73.45	74
sortCacheByGaussians	5.04	61.17	61.17	18
diagJTJ	41.60	71.13	71.13	90
sortX	4.45	15.15	60.30	36
applyJ	10.98	86.32	86.32	80
applyJT_p1	3.93	75.79	75.79	34
applyJT_p2	0.37	18.83	89.69	40
applyJT_p3	3.20	4.75	78.48	48

Table 5. **Profiler analysis of CUDA kernels.** We provide results measured on a RTX3090 GPU for building/resorting the gradient cache and running one PCG iteration on the MipNerf360 [4] "garden" scene with a batch size of one image.

C. Detailed Runtime Analysis

We provide additional analysis of the CUDA kernels by running the `NVIDIA Nsight Compute` profiler. We provide results in Tab. 5 measured on a RTX3090 GPU for building/resorting the gradient cache and running one PCG iteration on the MipNerf360 [4] "garden" scene with a batch size of one image. We add the suffixes `_p1, _p2, _p3` to signal the three kernels that we use to implement the respective operation (see Appendix A).

Comparing the runtime of the `buildCache` and `applyJT` kernels reveals the advantage of our proposed *per-pixel-per-splat* parallelization pattern. Both compute the identical Jacobian-vector product, but the `buildCache` kernel relies on the *per-pixel* parallelization pattern of the original 3DGS rasterizer [23]. However, we compute the result 4.8x faster using the gradient cache in the `applyJT` kernel. We also note that the compute and memory throughput as well as the register count of both kernels are roughly similar. This signals that our kernel implementation is equally efficient, i.e., there are no inherent drawbacks using our proposed GPU parallelization scheme.

D. Results Per Scene

We provide a per-scene breakdown of our main quantitative results against all baselines on all datasets. The comparisons against 3DGS [23] are in Tab. 6. The comparisons against DISTWAR [12] are in Tab. 7. The comparisons against gsplat [48] are in Tab. 8. The comparisons against Taming-3DGS [32] are in Tab. 9. Our method shows consistent acceleration of the optimization runtime on all scenes, while achieving the same quality.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [23]	treehill	0.631	22.44	0.330	1130
+ Ours	treehill	0.633	22.57	0.334	**836**
3DGS [23]	counter	0.905	28.96	0.202	1178
+ Ours	counter	0.904	28.89	0.206	**927**
3DGS [23]	stump	0.769	26.56	0.217	1234
+ Ours	stump	0.774	26.67	0.218	**895**
3DGS [23]	bonsai	0.939	31.99	0.206	1034
+ Ours	bonsai	0.938	31.84	0.208	**794**
3DGS [23]	bicycle	0.764	25.20	0.212	1563
+ Ours	bicycle	0.765	25.30	0.218	**1141**
3DGS [23]	kitchen	0.925	31.37	0.128	1389
+ Ours	kitchen	0.924	31.21	0.128	**1156**
3DGS [23]	flowers	0.602	21.49	0.340	1132
+ Ours	flowers	0.600	21.52	0.344	**819**
3DGS [23]	room	0.917	31.36	0.221	1210
+ Ours	room	0.916	31.10	0.224	**1004**
3DGS [23]	garden	0.862	27.23	0.109	1573
+ Ours	garden	0.863	27.30	0.110	**1175**

Method	Scene	Deep Blending [19]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [23]	playroom	0.901	29.90	0.247	1085
+ Ours	playroom	0.905	30.24	0.246	**861**
3DGS [23]	drjohnson	0.898	29.12	0.246	1359
+ Ours	drjohnson	0.901	29.23	0.248	**1040**

Method	Scene	Tanks & Temples [27]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
3DGS [23]	train	0.811	21.95	0.209	636
+ Ours	train	0.811	22.07	0.214	**579**
3DGS [23]	truck	0.877	25.40	0.148	837
+ Ours	truck	0.876	25.36	0.151	**747**

Table 6. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the 3DGS [23] baseline.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
DISTWAR [12]	treehill	0.633	22.47	0.327	898
+ Ours	treehill	0.635	22.54	0.332	**669**
DISTWAR [12]	counter	0.905	29.00	0.203	790
+ Ours	counter	0.904	28.91	0.205	**687**
DISTWAR [12]	stump	0.771	26.60	0.216	1017
+ Ours	stump	0.773	26.70	0.217	**760**
DISTWAR [12]	bonsai	0.939	32.13	0.206	677
+ Ours	bonsai	0.938	31.92	0.208	**578**
DISTWAR [12]	bicycle	0.763	25.19	0.212	1333
+ Ours	bicycle	0.764	25.26	0.218	**971**
DISTWAR [12]	kitchen	0.925	31.31	0.127	957
+ Ours	kitchen	0.924	31.14	0.128	**838**
DISTWAR [12]	flowers	0.602	21.45	0.340	884
+ Ours	flowers	0.596	21.48	0.348	**671**
DISTWAR [12]	room	0.916	31.41	0.221	803
+ Ours	room	0.916	31.40	0.224	**680**
DISTWAR [12]	garden	0.862	27.23	0.109	1338
+ Ours	garden	0.861	27.32	0.112	**1023**

Method	Scene	Deep Blending [19]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
DISTWAR [12]	playroom	0.900	29.81	0.247	729
+ Ours	playroom	0.905	30.24	0.246	**586**
DISTWAR [12]	drjohnson	0.898	29.13	0.247	953
+ Ours	drjohnson	0.901	29.13	0.249	**758**

Method	Scene	Tanks & Temples [27]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
DISTWAR [12]	train	0.812	22.05	0.209	504
+ Ours	train	0.810	22.10	0.216	**440**
DISTWAR [12]	truck	0.877	25.29	0.148	698
+ Ours	truck	0.877	25.28	0.150	**635**

Table 7. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the DISTWAR [12] baseline.

Method	Scene	MipNeRF-360 [4]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
gsplat [48]	treehill	0.634	22.44	0.324	973
+ Ours	treehill	0.635	22.54	0.332	**701**
gsplat [48]	counter	0.908	28.99	0.201	903
+ Ours	counter	0.904	28.91	0.205	**762**
gsplat [48]	stump	0.769	26.53	0.218	1097
+ Ours	stump	0.774	26.70	0.217	**793**
gsplat [48]	bonsai	0.937	31.95	0.208	783
+ Ours	bonsai	0.938	31.92	0.208	**646**
gsplat [48]	bicycle	0.765	25.21	0.206	1398
+ Ours	bicycle	0.765	25.26	0.218	**988**
gsplat [48]	kitchen	0.926	31.17	0.128	1086
+ Ours	kitchen	0.924	31.14	0.128	**921**
gsplat [48]	flowers	0.600	21.53	0.338	965
+ Ours	flowers	0.601	21.48	0.348	**709**
gsplat [48]	room	0.920	31.48	0.219	913
+ Ours	room	0.916	31.39	0.224	**753**
gsplat [48]	garden	0.869	27.48	0.105	1462
+ Ours	garden	0.861	27.32	0.112	**1085**

Method	Scene	Deep Blending [19]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
gsplat [48]	playroom	0.907	29.89	0.248	799
+ Ours	playroom	0.904	30.90	0.247	**626**
gsplat [48]	drjohnson	0.901	29.16	0.244	1040
+ Ours	drjohnson	0.901	29.07	0.251	**805**

Method	Scene	Tanks & Temples [27]			
		SSIM↑	PSNR↑	LPIPS↓	Time (s)
gsplat [48]	train	0.811	21.64	0.209	558
+ Ours	train	0.809	22.09	0.216	**381**
gsplat [48]	truck	0.880	25.35	0.149	735
+ Ours	truck	0.877	25.28	0.150	**447**

Table 8. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the gsplat [48] baseline.

Method	Scene	MipNeRF-360 [4]				
		SSIM↑	PSNR↑	LPIPS↓	Time (s)	#G (M)
Taming [32]	treehill	0.625	23.00	0.385	479	0.78
+ Ours	treehill	0.623	23.03	0.332	**381**	0.78
Taming [32]	counter	0.896	28.59	0.223	646	0.31
+ Ours	counter	0.894	28.51	0.205	**537**	0.31
Taming [32]	stump	0.734	25.96	0.292	405	0.48
+ Ours	stump	0.733	25.98	0.217	**315**	0.48
Taming [32]	bonsai	0.934	31.73	0.221	634	0.41
+ Ours	bonsai	0.932	31.64	0.208	**504**	0.41
Taming [32]	bicycle	0.716	24.78	0.295	485	0.81
+ Ours	bicycle	0.709	24.75	0.218	**376**	0.81
Taming [32]	kitchen	0.918	30.85	0.141	722	0.48
+ Ours	kitchen	0.918	30.84	0.128	**597**	0.48
Taming [32]	flowers	0.554	21.00	0.407	465	0.57
+ Ours	flowers	0.549	20.99	0.348	**365**	0.57
Taming [32]	room	0.906	31.12	0.251	621	0.22
+ Ours	room	0.906	31.21	0.224	**521**	0.22
Taming [32]	garden	0.852	27.24	0.128	638	1.90
+ Ours	garden	0.852	27.25	0.112	**483**	1.90

Method	Scene	Deep Blending [19]				
		SSIM↑	PSNR↑	LPIPS↓	Time (s)	#G (M)
Taming [32]	playroom	0.900	30.29	0.278	419	0.18
+ Ours	playroom	0.902	30.41	0.280	**324**	0.18
Taming [32]	drjohnson	0.899	29.38	0.269	475	0.40
+ Ours	drjohnson	0.899	29.40	0.271	**370**	0.40

Method	Scene	Tanks & Temples [27]				
		SSIM↑	PSNR↑	LPIPS↓	Time (s)	#G (M)
Taming [32]	train	0.815	22.18	0.205	411	0.36
+ Ours	train	0.802	22.28	0.241	**328**	0.36
Taming [32]	truck	0.879	25.40	0.146	514	0.27
+ Ours	truck	0.862	25.16	0.178	**292**	0.27

Table 9. **Quantitative comparison of our method and baselines.** We show the per-scene breakdown of all metrics against the Taming-3DGS [32] baseline. Additionally, we include the number of Gaussians in millions (#G (M)) that we obtained using the default hyperparameters for "budgeting".